Въведение в езика C++ (продължение)

Любомир Чорбаджиев
Технологическо училище “Електронни системи”
Технически университет, София
lchorbadjiev@elsys-bg.org

Revision : 1.9

27 септември 2004 г.

Указатели

- За даден тип T, типът T* е **указател към** T. С други думи, променливите от тип T* съдържат адрес на обект от типа T.

```
int a=42;
int* pa=&a;
```

- Основната операция, която се изпълнява върху указателите, е операцията *. Този оператор връща обекта, към който сочи указателя.

```
int a=42;
int* pa=&a;
int a1=*pa;
```

Масиви

- За даден тип `T`, типът `T[size]` е масив от `size` елемента от тип `T`. Елементите на масива се индексират (номерират) от `0` до `size-1`. Броят на елементите на масива трябва да бъде константен израз.

```
1 int b[3];
2 char* a[42];
```

- Многомерните масиви се дефинират като масиви от масиви.

```
1 int d2[10][10];
2 int d3[10][10][10];
```


Инициализация на масиви

- Началните стойности на елементите на даден масив, могат да се присвоят като се използва списък от стойности.

```
int v[]={1,2,3,4};
char ac[]={'a','b','c'};
```

- Когато масивът е деклариран без да е указан неговият размер, броят на елементите в масива може да се определи от размера на инициализиращия списък.
- Когато размерът на масива е указан явно, инициализирането на масива със списък, съдържащ повече елементи е грешка.

```
int v[2]={1,2,3}; // Грешка!
```

Инициализация на масиви

- Ако в списъка за инициализация на масива броят на елементите е по-малък от размера на масива, то на неинициализираните елементи се присвоява стойност по подразбиране. Следната инициализация

  ```
  int v[4]={1,2};
  ```

  е еквивалентна на

  ```
  int v[]={1,2,0,0};
  ```

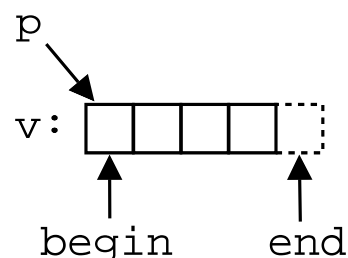
- За инициализирането на многомерни масиви се използва списък от списъци за инициализация.

  ```
  int v[][2]={{1,1},{2,2}};
  ```


Указатели към масиви

- Указателите и масивите са тясно свързани. Името на масива може да се използва като указател, сочещ към първият елемент на масива.

```
1 int v[]={1,2,3,4};
2 int* p=v;
3 int* begin=&v[0];
4 int* end=&v[4];
```



- Езикът гарантира, че стойността на указател, насочен след последният елемент на масива, е смислена. Тъй като този указател не сочи към елемент от масива, той не бива да бъде използван за четене на стойност или записване на стойност.

Достъп до елементите на масив

- Достъпът до елементите на масива се извършва или по индекс на елемента в масива, или чрез указател.

```
1 void fi(char v[]) {
2   for(int i=0;v[i]!=0;i++)
3     cout << v[i] << endl;
4 }
```

```
1 void fp(char v[]) {
2   for(char* p=v;*p!=0;p++)
3     cout << *p << endl;
4 }
```


Аритметика с указатели

- Изваждането на един указател от друг указател е дефинирано само в случай, че двата указателя сочат към елементи на един и същ масив.
- Резултатът от изваждането на два указателя е равен на броя елементи на масива, разположени между указателите.
- Към указателите може да се добавя и да се изважда цяло число.

Аритметика с указатели

- Когато към указател се добавя цяло число, резултатът ще бъде указател, отместен със съответният брой елементи към края на масива. Ако полученият указател не сочи към елемент на масива или към елемента след последния, резултатът не е дефиниран.
- Когато от указател се изважда цяло число, резултатът ще бъде указател, отместен със съответният брой елементи към началото на масива. Ако полученият указател не сочи към елемент на масива, то резултатът не е дефиниран.

Аритметика с указатели: пример

```
1 int v1[10];
2 int v2[10];
3
4 int i1=&v1[5]-&v1[3]; // резултатът е i1=2
5 int i2=&v1[5]-&v2[3]; // резултатът не е дефиниран
6
7 int* p1=v2+2; // p1=&v2[2]
8 int* p2=v2-2; // резултатът не е дефиниран
```

Константи

- В C++ е въведена концепцията за дефинирани от потребителя константи (**const**), чиято стойност не може да бъде променяна по време на изпълнение на програмата.
- За да се дефинира, че даден обект е константа, се използва ключовата дума **const**.

```
1 const int ans=42; // ans константа
2 const int v[]={1,2,3}; // всички v[i] са константи
```


Константи

- При дефинирането си, константите трябва задължително да бъдат инициализирани, тъй като след това тяхната стойност не може да се променя (в частност не може да им се присвоява стойност).

```
3 const int x; // грешка: няма инициализатор
```

- Когато обект е деклариран като **const**, неговата стойност не може да бъде променяна. Ключовата дума **const** модифицира типа, като ограничава възможната употреба на обекта.

```
4 ans=200; // грешка!
5 v[2]++; // грешка!
```

Указатели и константи

- При операциите с указатели участват два обекта – самият указател и обекта, към който сочи указателя.
- Когато ключовата дума **const** се постави пред дефиницията на указателя, това означава че константен е обекта към който сочи указателя.
- За де се декларира, че самият указател е константен, се използва ***const**, вместо *.

Указатели и константи: примери

```
1  char str1[]="hello";
2  char str2[]="hell";
3
4  const char* pc=str1;
5  pc[2]='a';  // грешка!
6  pc=str2;    // OK!
7
8  char *const cp=str1;
9  cp[2]='a';  // OK!
10 cp=str2;    // грешка!
11
12 const char *const cpc=str1;
13 cpc[2]='a'; // грешка!
14 cpc=str2;   // грешка!
```

Препратки (reference)

- За даден тип T, препратка към обект от тип T се обозначава с T&.
- Препратката (reference) е алтернативно име обект.

```
1 int i=1;
2 int& r=i;
3 r=2; // i=2
```

- Когато се дефинира препратка, тя задължително трябва да се инициализира.

```
1 int i=1;
2 int& r1=i;
3 extern int& r3;
4 int& r2;  // грешка: липсва инициализатор
5 int& r4=3;// грешка!
```


Препратки (reference)

- Независимо от формата на запис, операторите не се изпълняват върху препратката, а върху обекта към който тя препраща.

```
2 int i=0;
3 int& r=i;
4 r++;        // увеличава се i
5 int* p=&r; // p сочи към i
```

- Препратките най-често се използват като аргументи на функции, които трябва да променят стойността на предаденият обект.

```
1 void plus2(int& v) {v+=2;}
2 void f() {
3   int x=1;
4   plus2(x); // x=3;
5 }
```

Структури

- Структурата е обединение на елементи от (почти) произволен тип.

```
1 struct person {
2   char* name;
3   long int age;
4 };
```

- Името на структурата `person` се превръща в име на тип и могат да се дефинират променливи.

```
6 person somebody;
```

- Достъпът до членовете (полетата) на структурата се осъществява с използването на оператора . (точка).

```
7 somebody.name="ivan";
8 somebody.age=16;
```


Структури

- За инициализирането на структура се използва запис, подобен на инициализацията на масив.

```
9 person anybody={"pesho",18};
```

- Когато достъпът до структурата се извършва чрез указател, то членовете на структурата са достъпни чрез оператора ->.

```
11 void dump(person* ptr) {
12   cout << ptr->name << endl
13        << ptr->age << endl;
14 }
```

Област на действие (scope)

- **Област на действие** на една декларация се нарича частта от програмата, в която този декларация е валидна.
- В езика C++ са дефинирани три вида области на действие: глобална област на действие (file scope), локална област на действие (local scope) и област на действие на клас (class scope).

Глобална област на действие (file scope)

- **Глобална област на действие (file scope)** е частта от програмния код, която се намира извън дефинициите на функциите и класовете.
- Глобалната област на действие е най външната област на действие — тя огражда останалите области на действие.
- Декларация направена в глобалната област на действие е валидна до края на файла, в който е направена.

Локалната област на действие (local scope)

- **Локалната област на действие (local scope)** се свързва с дефиницията на дадена функция или с даденсъставен оператор (блок).
- Всеки блок дефинира своя локална област на действие, която се простира от началото до края на блока.
- Декларация направена в дадена локална област на действие е валидна до края на блока, в който е направена.
- Локалните области на действие могат да се влагат.
- Списъкът с аргументи на една функция принадлежи към нейната локална област на действие.

Област на действие на клас (class scope)

- Всеки клас се свързва с различна област на действие, която се нарича **област на действие на клас**.

Подробностите — друг път...

Област на действие: пример

- Променливи, които са дефинирани в глобалната област на действие се наричат **глобални**, а променливи дефинирани в дадена локална област на действие, се наричат **локални**.

```
1  int x;
2  void f() {
3    int x;   // скрива глобалната променлива x
4    x=1;
5    {
6      int x; // скрива външната локална променлива x
7      x=2;
8    }
9    x=3;
10 }
11 int* p=&x; // използва глобалната променлива x
```


Област на действие

- Глобалните променливи, които не са инициализирани явно, се инициализират по подразбиране с 0.
- Стойността на неинициализирана локална променлива е неопределена.

```
1  int i=0;
2  int j; // инициализира се по подразбиране с 0
3  int f() {
4    int i; // стойността е неопределена!
5    i++;
6    return i;
7  }
```

Област на действие

- Към скрита глобална променлива може да се обърнем, като използваме оператора за определяне на областта на действие – :: (четири точки):

```
int x;
void f() {
  int x=1;
  ::x=2;  // присвоява стойност на глобалната x
}
```